

Computational Thinking Patterns

Andri Ioannidou, Agentsheets Inc., Boulder, CO

Vicki Bennett, Communication Department, University of Colorado at Boulder

Alexander Repenning, Kyu Han Koh, Ashok Basawapatna, Computer Science Department, University of Colorado at Boulder

Paper presented at the 2011 Annual Meeting of the American Educational Research Association (AERA) in the Division C - Learning and Instruction / Section 7: Technology Research symposium "Merging Human Creativity and the Power of Technology: Computational Thinking in the K-12 Classroom"

Publication Date: April 8, 2011

Objectives

The iDREAMS project aims to reinvent Computer Science education in K-12 schools, by using game design and computational science for motivating and educating students through an approach we call Scalable Game Design, starting at the middle school level. In this paper we discuss the use of Computational Thinking Patterns as the basis for our Scalable Game Design curriculum and professional development and present results from measuring student learning outcomes using our unique Computational Thinking Pattern Analysis.

Background: Scalable Game Design project

The iDREAMS project¹ (Integrative Design-based Reform-oriented Educational Approach for Motivating Students), funded by the National Science Foundation under the ITEST program, investigates the potential impact on the IT workforce by stimulating interest in computer science at the middle school level through an approach called *Scalable Game Design* (Repenning & Ioannidou, 2008; Repenning, Ioannidou, et al., 2010; Repenning, Webb, & Ioannidou, 2010). The main goal of this project is to increase opportunities for computer science education in the regular curriculum and broaden participation, especially of women and underrepresented communities, by motivating and educating students about computer science through game design and computational science. The project serves a wide spectrum of communities: technology hub, urban/inner, rural, and remote Native American areas. An important goal is to develop local resources and capacity for sustainability through intensive teacher training and the involvement of local community and tribal colleges. The original target for the project was to deliver training for 28 teachers and 14 community/tribal college students through annual one- and two-week summer workshops and classroom instruction to 1120 students. Half way through the three-year project, we have already reached over 4000 students, 50 teachers and 30 community college students.

The Scalable Game Design approach is rooted in a theoretical framework combining models of motivation (Flow (Csikszentmihalyi, 1990)), competency frameworks and standards for technology (Computational Thinking (Wing, 2006), FIT (Committee on Information Technology Literacy, 1999), NETS (ISTE, 2007)), and computational thinking authoring tools (AgentSheets² (Ioannidou, Rader,

¹ <http://scalablegamedesign.cs.colorado.edu>

² <http://www.agentsheets.com>

Repenning, Lewis, & Cherry, 2003; Repenning & Ioannidou, 1997, 2004; Repenning, Ioannidou, & Ambach, 1998; Repenning, Webb, et al., 2010)).

- ***Models of Motivation:*** The Scalable Game Design curriculum balances design challenges and design skills to keep students in optimal flow (Csikszentmihalyi, 1990). Based on our experiences, the skills versus challenges space of Flow can index and relate a number of projects starting with a simple Frogger game all the way up to Sims-like games and scientific simulations. This can be achieved through various forms of scaffolding, such as explicit just-in-time instruction, social learning support from interactions with instructors and other students, and curricula designed in anticipation of the next challenge.
- ***Competency Frameworks:*** Computer science and game design are not just about learning how to code. Indeed, the common “computer science = programming” perspective is one of the main reasons why most students, and especially girls at the middle school level, have limited interest in computer science. Existing computer science curricula (ACM, 1993) work poorly in terms of attracting and retaining students (Computer Science Teachers Association, 2005). Unlike these curricula, the Scalable Game Design approach is based on Computational Thinking (Wing, 2006; Guzdial, 2008; Hambruch et al, 2009) and other frameworks such as the Fluency with Information Technology (FIT) framework (Committee on Information Technology Literacy, 1999) defined by National Academy of Sciences. These approaches do not base computational fluency entirely on programming skills. In contrast, they stress conceptual and design-related problem-solving skills.
- ***Computational Thinking Authoring Tools:*** The right kinds of tools are essential to enable even 10-year-old children to acquire Computational Thinking skills. We have been using and evolving the notion of computational thinking tools for over 15 years. Based on our experiences developing and using technology in schools settings, we claim that for systemic impact, a computational thinking tool used in K-12 must fulfill all these conditions (Repenning, Webb, and Ioannidou, 2010):
 - 1) has *low threshold*: a student can produce a working game quickly.
 - 2) has *high ceiling*: a student can make a real game that is playable and exhibits sophisticated behavior, e.g., complex AI.
 - 3) *scaffolds Flow*: the tool + curriculum provides stepping stones with managed skills and challenges to accompany the tool.
 - 4) *enables transfer*: tool + curriculum must work for both game design and subsequent computational science applications as well as support transfer between them. It should also facilitate transition to professional programming languages.
 - 5) *supports equity*: game design activities should be accessible and motivational across gender and ethnicity boundaries.
 - 6) is *systemic and sustainable*: the combination of the tool and curriculum can be used by all teachers to teach all students (e.g. include opportunities for teacher training and implementation support; align with standards and work in STEM content areas).

A number of authoring tools geared towards K-12 students, such as Scratch (Maloney, et al., 2004; Resnick, 2009), Alice (Conway, et al., 2000; Moskal, Lurie, & Cooper, 2004), GreenFoot (Poul & Michael, 2004), or NetLogo (Wilensky & Stroup, 1999), satisfy some but not all of the above criteria. AgentSheets (Figure 1), whose development was funded by the National Science Foundation, is an environment that encompasses all the computational thinking requirements in the above list. AgentSheets is an early incarnation of the now popular objects-first philosophy (Barnes & Kölling, 2006). In the spirit of scalability, the visual programming language featured in AgentSheets is highly accessible to students without any programming background. It lets novices create simple games such as Frogger in a few hours, yet it is powerful enough to let more experienced programmers create

sophisticated games such as Sims-like games using AI techniques (Repenning, 2006a, 2006b) and scientific simulations. AgentSheets also supports the transition to traditional programming by letting students render their visual programs directly into traditional programs. This versatility is essential for Scalable Game Design.

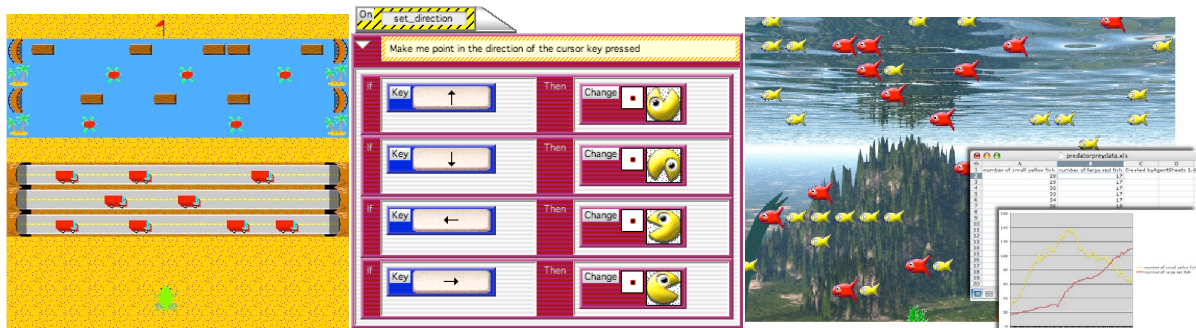


Figure 1: AgentSheets is a low-threshold, high-ceiling Computational Thinking tool that can be used to create simple games such as Frogger-like games (left) quickly, using an accessible drag-and-drop language (middle). It can also be used to create more sophisticated games and science simulations, such as ecosystem simulations (right).

As mentioned earlier, the goal of the iDREAMS project is to broaden participation in computer science by addressing both motivational and educational concerns. The *strategy* to maximize exposure includes the integration of computer science education into existing courses offered at middle schools. Middle schools are excellent places for increasing and broadening the participation in computer science, because those are critical years for students for reaching conclusions regarding their own skills and aptitudes (Gootman, 2007; Sears, 1995). Another reason to do this in the regular curriculum is to expose *all* students, including minorities and women, to computer science at a level of participation that no combination of extracurricular programs could. While many extracurricular programs have been successful, they only reach out to a small subset of children compared to the number of students enrolled in middle schools. After all, one should not forget that participation in extracurricular programs is based on self-selection. Students showing up at these events typically are already excited about information technology. Incorporating Scalable Game Design in the regular curriculum can reach the large majority of students who are skeptical towards IT or may not have the resources to participate in after-school programs.

The *Scalable Game Design curriculum* features required, elective, and transitional modules. In 6th grade a short module is integrated into an existing, and when possible required, course within the curriculum. In this context, we introduce all students to game design, teaching them to build a simple but complete game in one to two weeks. This serves mainly as a motivational module to get students to realize “I can do this!” Subsequently, typically in 7th and 8th grades, longer modules in elective courses allow students to increase their skill with more sophisticated games, including ones featuring Artificial Intelligence algorithms (Repenning, 2006a, 2006b), or transition to computational science and other STEM content. To enable that transition, our approach to teaching computer science goes beyond teaching programming. Instead, we use the notion of Computational Thinking to give students high-level problem-solving skills that are useful not just for game design with a specific programming language, but are more generally applicable to other areas, such as scientific modeling and more general problem solving.

The *evaluation* of the first year and a half of Scalable Game Design implementations in schools in Colorado, South Dakota, Wyoming, Texas, and Alaska revealed a number of exciting highlights:

- **Broadening participation:** We have had great results in terms of participation of girls and underrepresented communities - about 45% of our over 4000 participants so far are female; 56% are minority students.

- **Motivation:** Most students the Scalable Game Design modules independent of age/grade, gender, ethnicity, and location (inner city, rural, and remote Native American communities) are motivated to continue (about 61% of the girls; 71% of the boys; 71% of the white students, 69% of minority students). Keep in mind students are participating in this project through regular, formal courses. In other words, these are not the typical self-selected students who participate in after-school programs.
- **Learning outcomes:** Computational thinking is one of the main learning outcomes of this process. Every submitted student game gets analyzed automatically and a Computational Thinking Pattern Analysis graph gets created and inserted into the web page containing the game. We have also created a computational thinking survey that can assess the perception of computational thinking patterns outside the scope of computation, in video segments depicting real-world situations (Marshall, 2011). Our most exciting finding is that we begin to see early indications of transferable skills between game design and computational science that are measurable and begin to show that game design really does have strong educational consequences.

We consider the ability to measure and compute computational thinking through Computational Thinking Patterns, as an important breakthrough. Given the pending suspicion towards game design as an educational process by the general population, having measurable outcomes for motivation and learning helps to counter this criticism. The remainder of this paper presents our Computational Thinking Pattern Analysis framework that has produced early indicators of transfer from game design to computational science, after a brief introduction to Computational Thinking.

Theoretical framework: Computational Thinking Patterns

Ever since Jeannette Wing (2006) coined the term “computational thinking,” there has been a debate over providing a single definition. Despite the lack of a consistent definition, within the Computer Science community, a common understanding of the term emerged related to abstraction as a means for solving problems. According to P.J. Denning (2009), computational thinking is just a new term for a core concept, algorithmic thinking. Computer Science is not just about programming, but an entire way of thinking. The Computer Science community believes that learning to think in this way, to think computationally, could promote new perspectives and solutions to problems in many other disciplines (Wenger, 1998).

Among computer science articles (Lu & Fletcher, 2009; Orr, 2009; Qin, 2009; Qualls & Sherrell, 2010), the most common characteristics of computational thinking are abstraction and problem solving. Recently, the International Society for Technology in Education (ISTE) in collaboration with the Computer Science Teachers Association (CSTA) published a definitional list of computational thinking characteristics (ISTE, 2011). These include, but are not limited to:

- 1) Formulating problems for use with a computer to facilitate the solution;
- 2) Logically organizing & analyzing data;
- 3) Representing data through abstractions;
- 4) Automating solutions through algorithmic processes;
- 5) Identifying, analyzing and implementing possible solutions, as the most efficient & effective combination of steps and resources; and
- 6) Generalizing and transferring this process to variety of problem areas

The ability to think about problems in a more abstract manner is essential to students’ ability to see more solution opportunities. Thinking in this abstract manner, or computationally, is a way of accessing solutions that are usually outside a student’s normal area of expertise, or schemata. According to Anderson, Spiro and Anderson (1978), “to interpret a particular situation in terms of a schema is to match

the elements in the situation with the generic characterizations in the schematic knowledge structure.” Learning to think computationally or to problem-solve through abstraction is the ability to eliminate details from a given situation in order to find a solution that might not be forthcoming under other circumstances (Sarjoughian & Zeigler, 1996). While computational thinking is mostly referred to as the computer science approach to problem-solving through abstraction, the use of computational thinking in other disciplines can promote creativity (Bennett, Koh & Repenning, 2011) and innovation through abstractly conceptualizing the question or challenge. The more frequently students use computational thinking, the better they will become at finding alternative, unique and inventive solutions to complex problems in various domains.

In absence of a commonly accepted definition for computational thinking at the beginning of the Scalable Game Design project, we begun exploring the issue with an *expectation* set forth by an instructional technology director in a district implementing the curriculum. He wanted to be able to walk up to any student that had engaged in game design and say “Now that you made Space Invaders, can you make a science simulation?” Essentially he was asking for evidence for transfer from game design to computational science and other STEM areas. We therefore needed a way to consistently track student progress during implementation and analyze student artifacts upon completion to determine if there was any evidence of transfer. For the purpose of Scalable Game Design, computational thinking is a way of thinking that encompasses the use of *Computational Thinking Patterns* for building games and simulations.

Phenomenology: the origin of Computational Thinking Patterns

The origin of Computational Thinking Patterns goes back to the exploration of computational thinking transfer between different application domains, and more specifically between game design and science simulations, which could be explored at various levels. The simplest level of investigation would be the program level at which students express computational thinking ideas through writing code. There are many different programming languages, exhibiting different degrees of accessibility that could be employed. Let us consider a programming language such as Python, which certainly could, and indeed has been, used to create applications such as games and science simulations. The study of transfer could investigate the use of language statements such as the IF, THEN, ELSE, or LOOP statements found in many programming languages. Finding similar use of these statements in game and science simulations could be considered an indicator of low-level transfer of certain computer science skills. However, as mentioned above, most researchers investigating the notion of computational thinking do agree that computational thinking is not identical to programming. Consequently, analysis of computational thinking may have to take place at a higher level that is programming-language independent. For computational thinking, it is not of the outmost relevance *how* something is expressed (e.g., programmed differently in Flash, Java, or C++) but of *what* computational idea is expressed. In general, to find the intention behind a program is a difficult problem. However, for the relatively large universe of programs controlling some kind of object on a screen – which is common to most games and simulations – one could try to analyze object interactions. What kinds of behaviors are objects exhibiting? Are they static? Are they moving around? If they are moving around are they moving in some straight direction or are they wandering around randomly? If there is more than just one object, how are they, if at all, interacting with each other? Are objects colliding, pulling each other, or tracking each other? How do objects interact with users? Are users controlling object through a user interface? For instance, are they using cursor keys to move an object in orthogonal directions? Are they using a mouse or perhaps a gesture interface to control objects?

In science, *phenomenology* is used to describe empirical observations of occurrences. This is useful for capturing object interaction. Phenomenology can be employed to describe a wide variety of phenomena including mechanical phenomena and scientific phenomena. For instance, a mechanical phenomenon can be the study of the motion of objects. In his seminal work on the perception of causality, Michotte (1963) has shown that people interpretation of mechanical phenomena is not a cognitive process, but a perceptual

one. Michotte explored a range of mechanical phenomena between abstract objects including objects colliding and launching each other. We have adopted and extended the set of phenomena suggested by Michotte to build a basis for a set of *computational thinking patterns* relevant to game design as well as to science simulations (Figure 2).

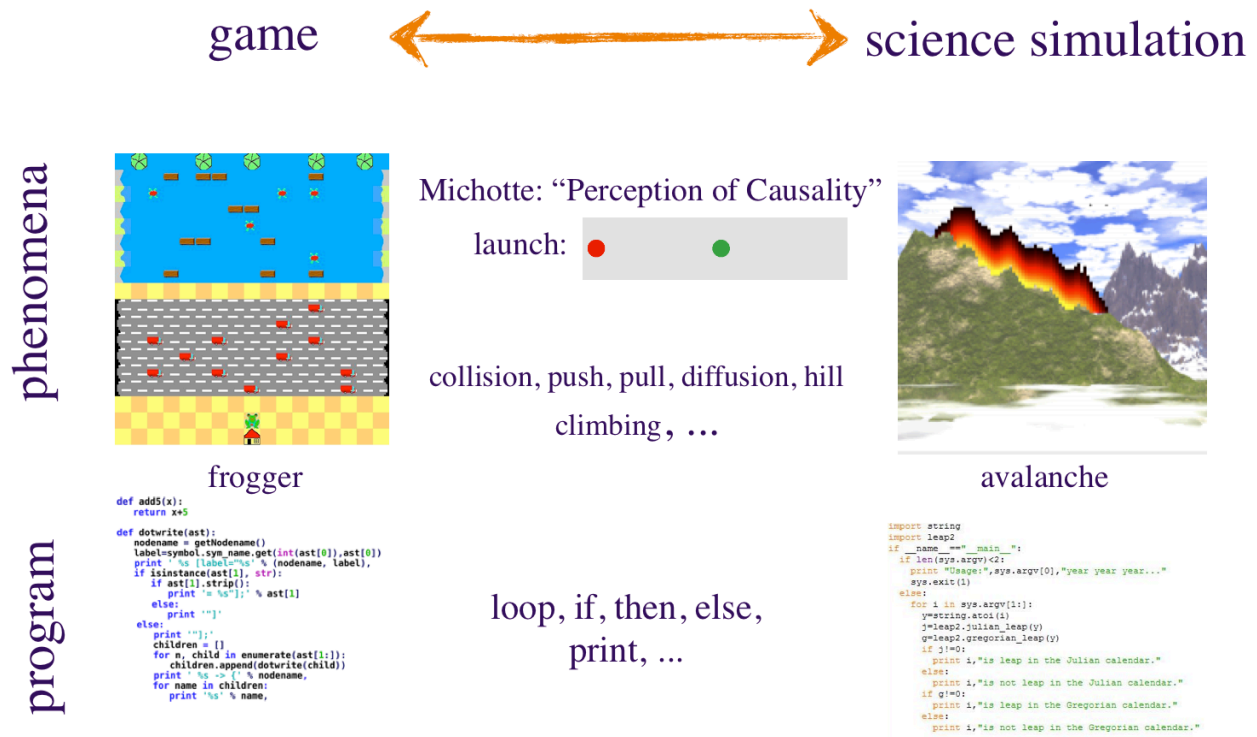


Figure 2: Phenomena is the right level for Computational Thinking Patterns for Scalable Game Design

Using this phenomenological analysis for thousands of games and simulations developed by AgentSheets users over the years, we created a list of basic and advanced Computational Thinking Patterns, including:

- **Collision:** deals with the event wherein two agents run into each other. E.g. in Frogger: frog meets truck; in a particle simulation: atoms collide with one another sometimes bonding together.
- **Push:** one object is pushing another. E.g. in Sokoban: the warehouse keeper pushes the boxes around.
- **Pull:** one object pulls another or many objects. E.g. a locomotive may be pulling a large number of railroad cars
- **Transport:** one object carries another object. E.g. in Frogger: logs and turtles transport frogs; in a blood simulation, red blood cells carry iron; in a factory simulation, a conveyor belt transports components.
- **Generation:** multiple objects get generated from a source. E.g. in Space Invaders: defenders shoot rockets; in Frogger, tunnels create cars
- **Absorption:** the opposite of the generation pattern. An object absorbs a stream of other objects. E.g. in Frogger: tunnel absorbs cars
- **Choreography:** in Space Invaders: mothership coordinates alien ships movement and descent
- **Diffusion:** spread of physical particles (scent, heat, electrons) or conceptual elements (rumors, ideas). E.g. in Pac-Man, the presence of Pac-Man is diffused through the game for the ghosts to follow. In a conservation simulation, heat is diffused from sources in a house.

- **Path Finding / Seeking:** objects move towards some objective in a complex environment like a maze. E.g. in The Sims: people finding food; in an ecosystem simulation, predators follow prey.
- **Collaborative Diffusion:** multiple objects collaborate or compete with each other. E.g. in a soccer game: players collaborate with their teammates and compete with the players of the opposing team.
- **Multiple Needs:** Maslow's model of human motivation. Objects or characters pursue a variety of needs, like food, entertainment, and social interaction in the Sims game. Or subsumption architectures in robotics (decomposing complicated intelligent behavior into many "simple" behaviors)

Initially, computational thinking patterns were intended as a theoretical framework to conceptualize object interactions. This framework allowed students to dissect game descriptions and to describe computational thinking pattern found. Each computational thinking pattern would not only include a description of the phenomenon but also instructions on how to operationalize the pattern. For instance, how would one program an object collision in a certain programming environment? These operationalized descriptions range from concrete instructions such as code (e.g., how to implement a collision in Flash) to a more theoretical treatment such as a UML sequence diagram representing object interactions in a programming language-agnostic way.

This gradually expanding set of computational thinking patterns was first used in a number object oriented design and game design courses offered at the undergraduate level in computer science. Over time, as the list of patterns as well as their explanations got refined to the point that computational thinking patterns also proved accessible to much younger and significantly less experienced audiences including middle school students with no computer science background.

More recently, we have worked towards the ability to *compute computational thinking*. Could we use computational means to inspect artifact created such as games and simulations and actually detect computational thinking patterns?

Computing Computational Thinking: Computational Thinking Pattern Analysis

For computing Computational Thinking and automatically measuring learning outcomes in the context of the Scalable Game Design project, we built a cyberlearning infrastructure (National Science Foundation, 2008), called Scalable Game Design Arcade, for collecting and analyzing games and simulations created by students using AgentSheets in the participating schools. The Arcade consists of a main page displaying the most recently submitted games, most downloaded games, and most played games (Figure 3, left), an assignments gallery showing all the games/simulations submitted by students in a single class (Figure 3, right), and individual submission pages (Figure 4). The individual submission page includes a screenshot the game/simulation, links for playing and downloading the game/simulation, ratings and comments by other users, and the Computational Thinking Pattern Analysis of the game.

The Computational Thinking Pattern Analysis (CTPA) tools incorporated into the Arcade analyze the game/simulation upon submission and display the following:

- **CTPA Graph:** The submitted game is analyzed in terms of the Computational Thinking patterns it implements and displays the results on a graph (Figure 5, brown spider graph). If there is a tutorial for a submitted game/simulation, then the CTPA graph also depicts the computational thinking patterns of the tutorial (Figure 5, green spider graph). That is an indication of how the submitted game diverges from the tutorial implementation of the game, if one is available.
- **Similarity Scores and Matrix:** The analysis also yields structural similarity results for games. The submitted game is analyzed for similarity to four tutorial games (Frogger, Sokoban, Space Invaders,

and The Sims) and also to every other game submitted to the arcade. The first analysis yields a numerical score (Figure 4, middle right), and the second a similarity matrix (Figure 4, bottom right). The Similarity Score Matrix shows 25 projects sharing the most similar in programming structure submissions in the Arcade. Each is displayed with a direct link to the project and a similarity score depicting how much two given games are similar in their programming structure.



Figure 3: The main page of the Scalable Game Design Arcade (left) and an assignment page assignment (right) with all the student submissions for that assignment.

Technically speaking, the Computational Thinking Pattern Analysis is designed to evaluate the semantic meaning of a submitted game/simulation to the Arcade. A technique based on Latent Semantic Analysis (Landauer, 1998) is applied to CTPA to analyze a given submission in terms of computational thinking patterns. CTPA compares a given game/simulation with nine pre-defined canonical computational thinking patterns: cursor control, generation, absorption, collision, transportation, push, pull, diffusion, and hill climbing. CTPA compares the given game/simulation with each canonical Computational Thinking pattern (Figure 5, left) to produce the corresponding values in the CTPA graph (Figure 5, right). The calculated value represents how each Computational Thinking Pattern is similar to a given game/simulation. Higher value means higher similarity. If a specific Computational Thinking pattern is used a lot in the implementation of the game/simulation, then the similarity between that pattern and the game/simulation is increased. Therefore, more dominant Computational Thinking patterns in a game/simulation yield a higher value in the CTPA calculation.

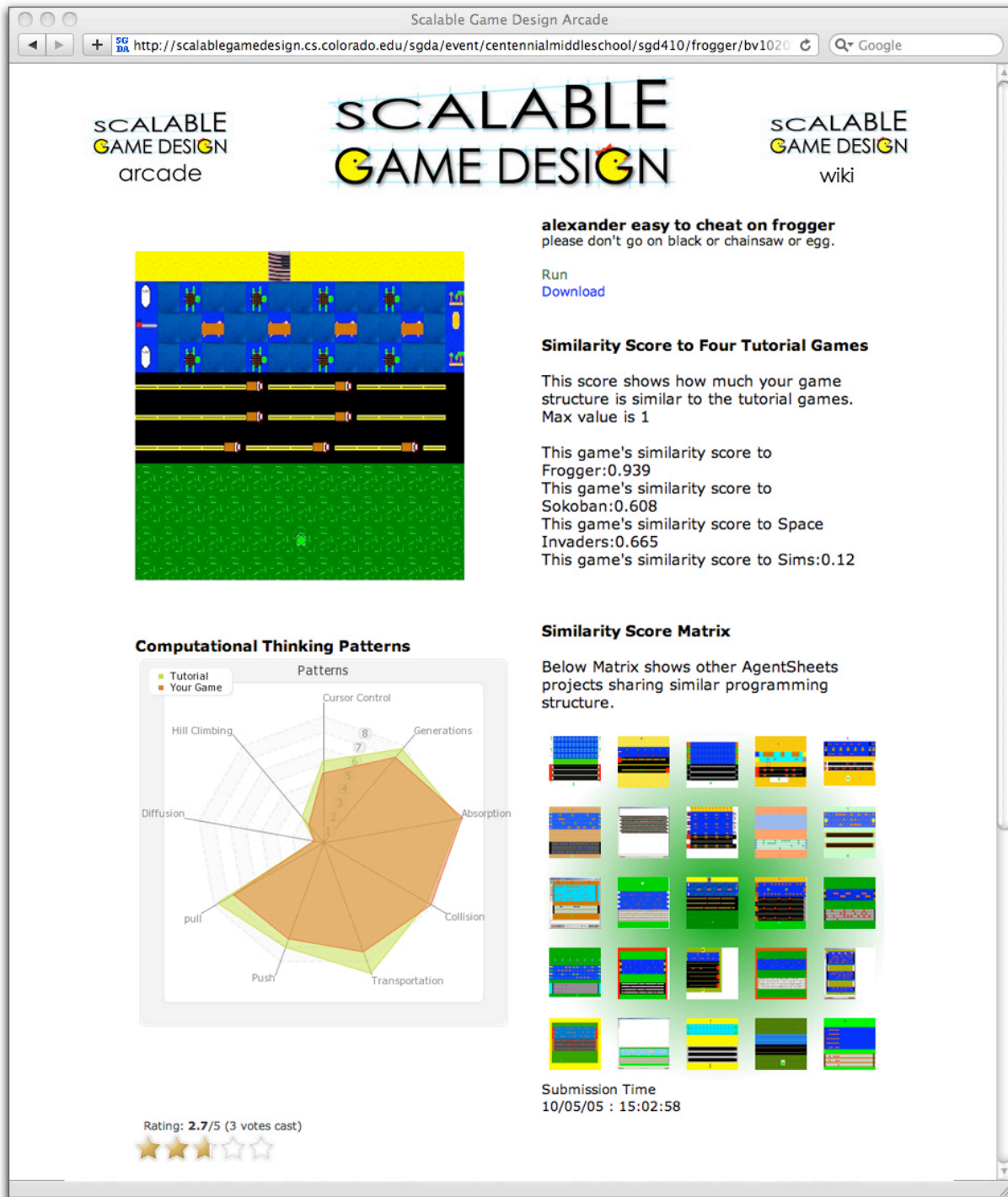


Figure 4: An individual game submission page for a Frogger game, with the screenshot of the game (top left), Run and Download links (top right), the game's similarity score compared to four tutorial games (middle right), the CTPA graph analyzing the game in terms of the Computational Thinking Patterns it contains and its similarity to the tutorial implementation of the Frogger game (bottom left), and a similarity score matrix showing similar games to the submitted one (bottom right).

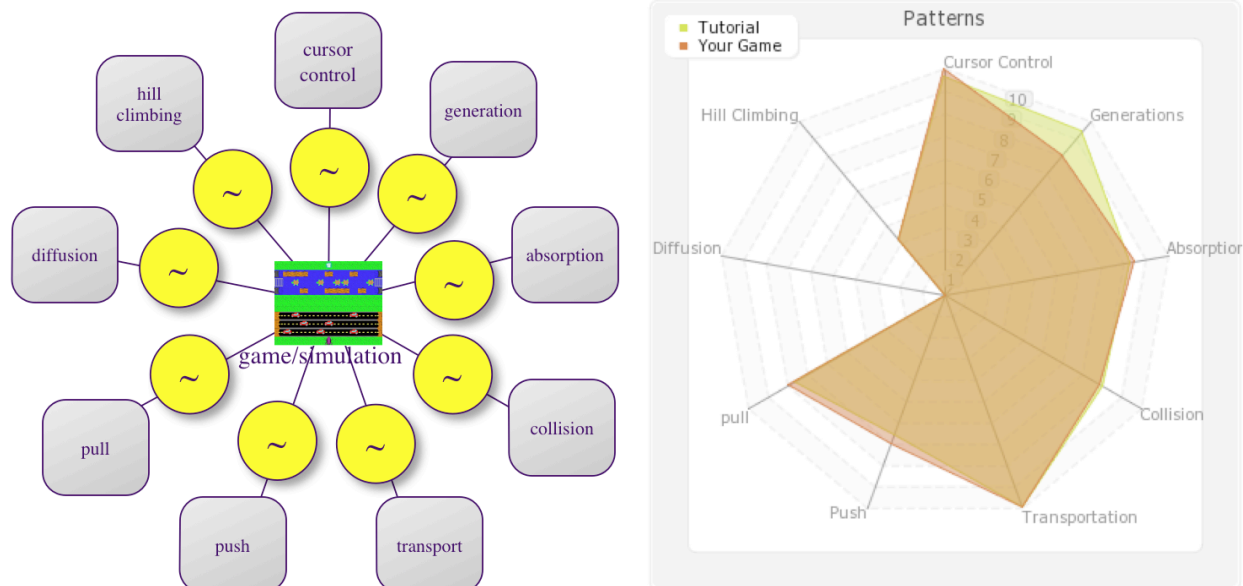


Figure 5: Using LSA-inspired similarity (denoted as ~ on the left), a game submitted to the Scalable Game Design Arcade gets compared to nine canonical Computational Thinking Patterns. A CTPA graph showing the similarity values for each pattern is produced (right).

The shape of a CTPA graph is similar for similar games. That is, if students implement the same game, then shapes of the CTPA graphs for that game would be similar to each other regardless students' programming style. For example, Figure 6 illustrates two Centipede games that were programmed by two different students. The two Centipede games are structurally different because of the different embellishments that each student put in the game (e.g. one includes scoring, the other does not), but semantically they are similar to each other. Consequently, two different Centipede games look similar on the CTPA graph.

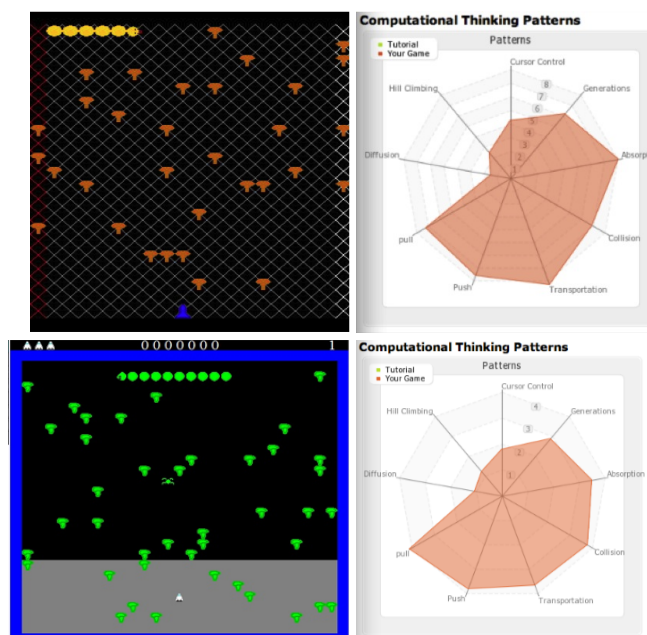


Figure 6: Two structurally different Centipede games produced by different students, one with many more agents and behaviors (such as scoring) (bottom) than the other (top), with their corresponding and similar in shape CTPA graphs (Koh, et al., 2010).

Significance: Measurable Learning Outcomes and Early Indicators of Transfer

By automatically breaking down complex programs into constituent parts that specifically enable students to create not just games, but also simulations, CTPA is a first step for measuring learning outcomes in the context of Computational Thinking. Other systems use Web 2.0 technology for enabling students to collaborate with each other, inspire each other, and get some educational benefits from collaborative learning exist (e.g. submitting Scratch games to the online repository), but there is no mechanism to compare each other's artifacts in terms of what skills were acquired in the process. CTPA is already usable for analyzing student artifacts built in AgentSheets in terms of computational thinking skills. As such, teachers and students can use CTPA for assessing whether a desired learning goal was met. The ability to automatically detect computational thinking patterns by analyzing student creations can give educators greater insight into what concepts the students understand and what concepts still need to be taught to them. This has been used for developing Scalable Game Design curriculum that supports a more natural and fluid progression of student exposure to Computational Thinking, in a way that balances skills and challenges (Csikszentmihalyi, 1990).

However, the really intriguing aspect of the Computational Thinking Pattern Analysis lies within the idea that Computational Thinking Patterns are common, shared elements between games and other STEM disciplines and that are identifiable and measurable. If students can build games using Computational Thinking Patterns, it might be possible that they can apply these same patterns to the implementation of science simulations that use those patterns. And if we can measure these patterns in the resulting artifacts, we can begin to show transfer of skills between domains. The patterns a student uses to implement games can be almost identical to the patterns they would use to implement an analogous science simulation. The only difference is that the agents who perform this interaction have changed. For example in Frogger, the frog is transported by a log that is floating on a river. The implementation of this log transporting the frog is the same way one might implement a red blood cell transporting iron in a blood flow science simulation. We have already used CTPA to show early indications of transfer (Koh, Basawapatna, Bennett, & Repenning, 2010). In Figure 7, three AgentSheets projects were made by a single student in a Scalable Game Design class. In chronological order, the student implemented Sokoban and Sims games as well as an ecosystem science simulation. The shape of the CTPA graph of this student's science simulation is similar to a combined CTPA graph of the Sokoban and Sims games that the student made before creating that science simulation. This can be interpreted that this student used the programming skills learned from making Sokoban and Sims to create a science simulation.

These are early indicators for transfer between game design and science simulation design. While further work is needed to discern correlation from causation and investigate the role of teachers in scaffolding concepts to be transferred, having measurable indications of transfer is an important step for computer science in general and computer science education in particular. People often assume that any programming experience leads to transfer, but transfer claims should not be overzealous. As Roy Pea points out in his critique of the transfer claims of LOGO:

This idea--that programming will provide exercise for the highest mental faculties, and that the cognitive development thus assured for programming will generalize or transfer to other content areas in the child's life--is a great hope. Many elegant analyses offer reasons for this hope, although there is an important sense in which the arguments ring like the overzealous prescriptions for studying Latin in Victorian times [] (Pea, 1983, p. 2).

The specific nature of the units of transfer in CTPA, namely the Computational Thinking Patterns, and the fact that we can identify and analyze them make it possible to begin to show that transfer is actually happening. This makes CTPA unique tool in that it is among the first tools that are able to automatically compute Computational Thinking, measure learning outcomes, and provide *initial indicators of transfer* in student-created artifacts.

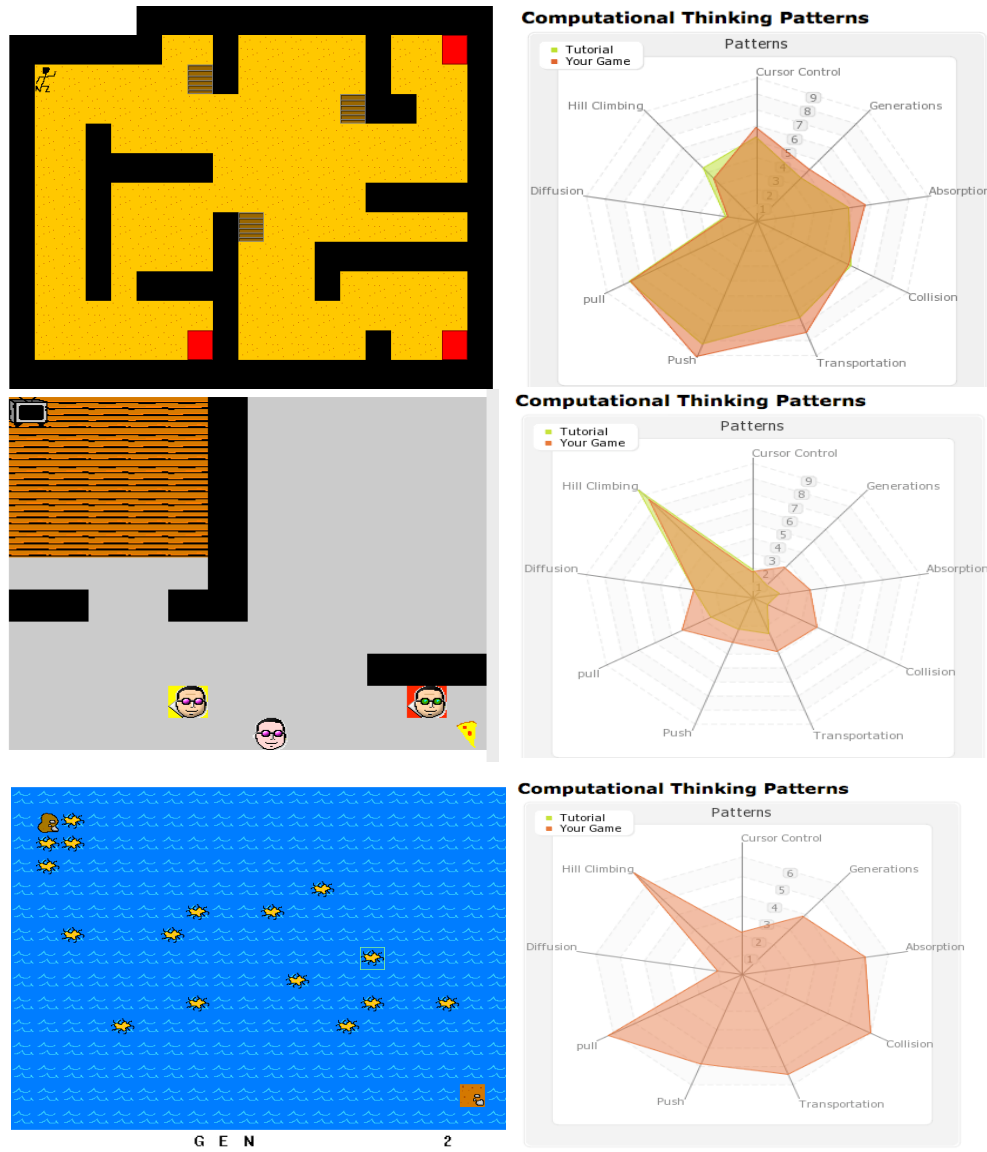


Figure 7: A Sokoban game screenshot and its CTPA graph (top). A Sims game made by the same student and its CTPA graph (center). A science simulation based on Chaos theory made by the same student and its corresponding CTPA graph (bottom). The combined CTPA graph of Sokoban and Sims are very similar to the CTPA graph of the science simulation, offering some early indications of transfer of skills from game design to computational science (Koh et al., 2010).

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant Numbers No. DLR-0833612 and IIP-0848962. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

- Anderson, R.C., Spiro, R.J. & Anderson, M.C. (1978). Schemata as scaffolding for the representation of information in connected discourse. *American Educational Research Journal*, 15, 433-440.
- Association of Computing Machinery, Corporate Pre-College Task Force Committee of the Educational Board of the ACM (1993). ACM model high school computer science curriculum. *Communications of the ACM*, 36(5), 87-90.
- Barnes, D. J., & Kölling, M. (2006). *Objects First with Java: A Practical Introduction using BlueJ* (Third ed.): Pearson Education / Prentice Hall.
- Bennett, V., Koh, K. & Repenning, A. (2011). CS Education Re-Kindles Creativity In Public Schools. To be presented at annual conference proceedings of *ITCSE 2011* in Germany.
- Committee on Information Technology Literacy: Lawrence Snyder, A. V. A., Marcia Linn, Arnold Packer, Allen Tucker, Jeffrey Ullman, Andries Van Dam (1999). *Being Fluent with Information Technology*. Washington, D.C.: National Academy Press.
- Computer Science Teachers Association (CSTA), Board of Directors (2005). *Achieving Change: The CSTA Strategic Plan*.
- Conway, M., Audia, S., Burnette, T., Cosgrove, D., Christiansen, K., Deline, R., et al. (2000). *Alice: Lessons Learned from Building a 3D System For Novices*. Paper presented at the CHI 2000 Conference on Human Factors in Computing Systems, The Hague, Netherlands.
- Csikszentmihalyi, M. (1990). *Flow: The Psychology of Optimal Experience*. New York: Harper Collins Publishers.
- Denning, P.J. (2009). Beyond computational thinking. *CACM*, 1, 1-6.
- Gootman, E. (2007, March 17, 2007). The Critical Years: For Teachers, Middle School Is Test of Wills. *New York Times*, from <http://www.nytimes.com/2007/03/17/education/17middle.html?ex=1331784000&en=a041618672b57230&ei=5088&partner=rssnyt&emc=rss>
- Guzdial, M. (2008). Education: Paving the way for computational thinking. *Communications of the ACM*, 51, 25-27.
- Hambrusch, S., Hoffman, C., Korb, J.T., Haugan, M. & Hosking, A.L. (2009). A multidisciplinary approach towards computational thinking for Science majors. *SIGCSE'09*, 183-187.
- ISTE International Society for Technology in Education (2007). *National Educational Technology Standards for Students (NETS)* (2nd ed.) Available at <http://www.iste.org/standards/nets-for-students.aspx>
- Ioannidou, A., Rader, C., Repenning, A., Lewis, C., & Cherry, G. (2003). Making Constructionism Work in the Classroom. *International Journal of Computers for Mathematical Learning*, 8, 63-108.
- Koh, K. H., Basawapatna, A., Bennett, V., & Repenning, A. (2010). *Towards the Automatic Recognition of Computational Thinking for Adaptive Visual Language Learning*. Paper presented at the Proceedings of the 2010 Conference on Visual Languages and Human Centric Computing (VL/HCC 2010), Madrid, Spain.
- Lu, J.J. & Fletcher, G.H.L.(2009). Thinking about computational thinking. *SIGCSE'09*, 260-264.
- Maloney, J., Burd, L., Kafai, Y., Rusk, N., Silverman, B., & Resnick, M. (2004). *Scratch: A Sneak Preview*. Paper presented at the Second International Conference on Creating, Connecting, and Collaborating through Computing, Kyoto, Japan.

- Marshall, K. (2011). *Was that CT? Assessing Computational Thinking Patterns through Video-Based Prompts*. Paper presented at the AERA Annual Meeting.
- Michotte, A. (1963). *The Perception of Causality* (T. R. Miles, Trans.). London: Methuen & Co. Ltd.
- Moskal, B., Lurie, D., & Cooper, S. (2004). *Evaluating the effectiveness of a new instructional approach*. Paper presented at the Proceedings of the 35th SIGCSE technical symposium on Computer science education.
- National Science Foundation, Task Force on Cyberlearning (2008), *Fostering Learning in the Networked World: The Cyberlearning Opportunity and Challenge. A 21st Century Agenda for the National Science Foundation*, Report available at <http://www.nsf.gov/pubs/2008/nsf08204/nsf08204.pdf>.
- Orr, G. (2009). Computational thinking through programming and algorithmic art. *SIGGRAPH2009*, New Orleans, LA.
- Pea, R. (1983). *LOGO Programming and Problem Solving*. Paper presented at symposium of the Annual Meeting of the American Educational Research Association (AERA), "Chameleon in the Classroom: Developing Roles for Computers," Montreal, Canada, April 1983.
- Poul, H., & Michael, K. (2004). *Greenfoot: Combining Object Visualisation with Interaction*. Paper presented at the Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications.
- Qin, H. (2009). Teaching computational thinking through bioinformatics to Biology students. *SIGCSE'09*, 188-191.
- Qualls, J.A. & Sherrell, L.B. (2010). Why computational thinking should be integrated into the curriculum. *JCSC*, 25, 66-71.
- Repenning, A. (2006a). *Collaborative Diffusion: Programming Antiojects*. Paper presented at the OOPSLA 2006, ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications, Portland, Oregon.
- Repenning, A. (2006b). *Excuse me, I need better AI!: employing collaborative diffusion to make game AI child's play*. Paper presented at the ACM SIGGRAPH symposium on Videogames, Boston, Massachusetts.
- Repenning, A., & Ioannidou, A. (1997). *Behavior Processors: Layers between End-Users and Java Virtual Machines*. Paper presented at the Proceedings of the 1997 IEEE Symposium of Visual Languages, Capri, Italy.
- Repenning, A., & Ioannidou, A. (2004). Agent-Based End-User Development. *Communications of the ACM*, 47(9), 43-46.
- Repenning, A., & Ioannidou, A. (2008). *Broadening Participation through Scalable Game Design*. Paper presented at the Proceedings of the ACM Special Interest Group on Computer Science Education Conference, (SIGCSE 2008), Portland, Oregon USA.
- Repenning, A., Ioannidou, A., & Ambach, J. (1998). Learn to Communicate and Communicate to Learn. *Journal of Interactive Media in Education*, 98(7).
- Repenning, A., Ioannidou, A., Webb, D., Keyser, D., MacGillivray, H., Marshall, K. S., et al. (2010). *Teaching Computational Thinking through the Scalable Game Design Curriculum*. Paper presented at the Paper presented at the 2010 American Educational Research Association (AERA) Annual Meeting, Denver, Colorado.
- Repenning, A., Webb, D., & Ioannidou, A. (2010). *Scalable Game Design and the Development of a Checklist for Getting Computational Thinking into Public Schools*. Paper presented at the To

- appear in the proceedings of the 2010 ACM Special Interest Group on Computer Science Education (SIGCSE) Conference, Milwaukee, WI.
- Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., Kafai, Y. (2009). Scratch: Programming for All. *Communications of the ACM*, November 2009.
- Sarjoughian, H.S. & Zeigler, B.P. (1996). Abstraction Mechanisms in discrete-event inductive modeling. *Proceedings of the 1996 Winter Simulation Conference*, 748-755.
- Sears, S. J. (1995). Career and Educational Planning in the Middle Level School. NASSP Bulletin, April 1995.
- Wenger, E. (1998). *Communities of Practice: Learning in Doing: Social, Cognitive and Computational Perspectives*. New York, NY: Cambridge University Press.
- Wilensky, U., & Stroup, W. (1999). *Learning through Participatory Simulations: Network-Based Design for Systems Learning in Classrooms*. Paper presented at the Computer Supported Collaborative Learning (CSCL '99), Stanford University, CA: December 12-15.
- Wing, J. M. (2006). Computational Thinking. *Communications of the ACM*, 49(3), 33-35.